To prove that ResizingArrayStackOfStrings is never less than one-quarter full and to show that for any ResizingArrayStackOfStrings client, the total cost of all stack operations divided by the number of operations is bounded by a constant, we can use the concepts of amortized analysis and geometric progression.

## Proof that the Array is Never Less Than One-Quarter Full

Consider a dynamic array that doubles its size when full and halves its size when it is one-quarter full. Let $n$ be the number of elements in the array, and let $N$ be the current capacity of the array.

1. **Doubling the Array Size:**
   - When the array is full ($n = N$), we double the size of the array to $2N$.
   - The array will be half full immediately after resizing because $n = N$ and the new capacity is $2N$.
2. **Halving the Array Size:**
   - When the array is one-quarter full ($n = N/4$), we halve the size of the array to $N/2$.
   - The array will be half full immediately after resizing because $n = N/4$ and the new capacity is $N/2$.

By this resizing strategy, the array is always between one-quarter and full capacity. Specifically, the array is never less than one-quarter full because we only halve the array when it is exactly one-quarter full, ensuring that the new size will be half full.

## Amortized Analysis of Stack Operations

To show that the total cost of all stack operations divided by the number of operations is bounded by a constant, we use amortized analysis.

1. **Cost of Operations:**
   - **Push Operation:** If the array is not full, the push operation takes $O(1)$ time. If the array is full, we need to resize the array, which takes $O(N)$ time, where $N$ is the current capacity. After resizing, the push operation takes $O(1)$ time.
   - **Pop Operation:** If the array is not one-quarter full, the pop operation takes $O(1)$ time. If the array is one-quarter full, we need to resize the array, which takes $O(N)$ time. After resizing, the pop operation takes $O(1)$ time.
2. **Amortized Cost:**
   - Consider a sequence of $m$ operations. Each time the array doubles, it can accommodate twice as many elements before needing to resize again.

- The total cost of resizing operations over $m$ operations can be expressed as a geometric series. If we double the size of the array, the cost of resizing is $O(1 + 2 + 4 + \ldots + N) = O(N)$, where $N$ is the final capacity.
- Since the array doubles in size, the number of resizing operations is $O(\log m)$. The total cost of all operations is $O(m + m/2 + m/4 + \ldots + 1) = O(m)$.

3. **Amortized Cost per Operation:**

- The total cost of $m$ operations is $O(m)$.
- The amortized cost per operation is $O(m)/m = O(1)$.

## Doubling the Array Size

When we double the size of an array, the cost of copying elements from the old array to the new array is proportional to the number of elements, $n$.

1. **Initial Setup:**

- Start with an array of capacity $1$.
- Each time the array is full, double its capacity.

2. **Cost Analysis:**

- **Insertion without resizing:** Takes $O(1)$ time.
- **Insertion with resizing:** Takes $O(n)$ time for resizing, plus $O(1)$ for the actual insertion.

3. **Amortized Cost Calculation:**

- Let's consider a sequence of $m$ insertions.
- Every time we double the array, the number of elements that need to be copied is the current size of the array.

For example:

- Insertions 1 to 1: No resizing needed, cost is $1$.
- Insertions 2 to 2: Resize from 1 to 2, cost is $1$ for copying, plus $1$ for insertion.
- Insertions 3 to 4: Resize from 2 to 4, cost is $2$ for copying, plus $1$ for insertion.
- Insertions 5 to 8: Resize from 4 to 8, cost is $4$ for copying, plus $1$ for insertion.
- ...

The cost of each resize operation can be represented as a geometric series:
$1 + 2 + 4 + 8 + \ldots + n$

The sum of this series is $2n - 1$, which is $O(n)$.

4. **Amortized Cost per Operation:**

- The total cost for $m$ insertions is $O(m)$ for the individual insertions plus $O(m)$ for the resizing operations.
- Therefore, the total cost for $m$ operations is $O(m) + O(m) = O(m)$.

- The amortized cost per operation is: $\frac{O(m)}{m} = O(1)$

### Why Not $O(\log m)$?

The factor of $O(\log m)$ might come up in discussions about the number of resizing operations, not the cost per operation. Specifically:

- **Number of Resizings:**
  - The array size doubles, so the number of times we need to resize the array for $m$ insertions is $O(\log m)$.
- **Cost of Resizing:**
  - Each resizing operation itself is $O(n)$, where $n$ is the current number of elements.
  - The total cost of all resizing operations over $m$ insertions is still $O(m)$.

The key point is that while the number of resizings is $O(\log m)$, the total cost of all operations, including resizings, divided by the number of operations, remains $O(1)$.

## Conclusion

The amortized cost of operations in a dynamically resizing array that doubles its size when full is $O(1)$. This is because the total cost of all operations, including the occasional expensive resizing operation, is linear with respect to the number of operations, and thus the average cost per operation is constant. The number of resizing operations grows logarithmically, but this does not affect the amortized cost per operation, which remains $O(1)$.